CS 5830 Homework 1

Instructor: Thomas Ristenpart
TAs: Andres Fabrega, Sanketh Menda, and Dhrumil Patel
Due: 21 September 2022

Submission and Grading

Submit your written answers to Gradescope and your code answers to CMSX. You must submit appropriate answers to the both submission sites for full credit.

Problems in this assignment are of two types:

1. *Attacking cryptography.* Given an incomplete template file, you need to fill in the missing pieces to mount an attack, e.g. retrieving a flag or forging a ciphertext.
2. *Talking cryptography.* Talk us through how you attacked a given cryptographic primitive or how you would hypothetically attack it.

The exact materials and the format you would need to submit them in for a given problem are indicated at the end of the problem.

Late Days

Homeworks are due on the due date by 11:59pm EST. You can use in total 3 late days throughout the semester.

Academic Integrity

Feel free to refer to external materials, but as usual please acknowledge them in your writeup in accordance with the Cornell Code of Academic Integrity.

Server Availability

To query the server, you'll use the same username and API key throughout this semester. Your username is your Cornell NetID (like `sm2289`) and you can find your assigned API key on CMSX, in the *Grading Comments & Requests* section of the placeholder assignment called `API_keys`.

We made a best effort to ensure the reliability of the challenge servers, but unfortunately, due to the nature of software and the internet, there may be disruptions. If you run into a disruption (the server is not responding, it is providing an error code, etc.), please do not panic, and reach out to the TAs (`@sanketh`, `@Andres`, or `@Dhrumil Patel`) on Slack. If this happens within 24 hours of the deadline, we will extend the deadline to accommodate for the inconvenience.

Verbosity for Written Answers

Generally, we prefer concise answers.

For a question like "Explain how you mount the attack?" we are expecting a high-level, but precise description of the attack in 3-10 sentences. Here is an example of a good answer.

> I started with an initial guess of the 16-byte all-zeros message. Then for each of the 16 bytes, I did X. Then did Y. Since the success of this is not guranteed, I repeated the process 20 times and aggregated. With the confidence, I did Z to recover the `FLAG`.

For a question like "Estimate the number of queries your attack issues?" we want you to think about how many queries you *really* need, and expect a high-level answer with ballpark numbers. Here is an example of a good answer.

> My attack does 5 queries initially for [purpose]. Then it does 1 queries per `FLAG` byte. In sum, for a flag of length N, it needs 1 + 5/N queries per `FLAG` byte.

Flag Notes

Each problem includes a description of the expected flag. For instance, it may say that it has 10 alphanumeric characters. Feel free to use that to restrict your search space, and to verify that you found the right flag (if you find something that meets the flag description, it is overwhelmingly likely that you found the right flag).

During grading, we will run your code against a different flag that nevertheless meets the description. So please make sure that you are not hardcoding assumptions regarding your specific flag.

Question 1. Stream Cipher Biases

Introduction. Suppose you run into your friend Alice from high school at the mall while buying shoes, and she tells you that she is now selling random streams for encryption. She claims that the streams are almost perfectly random, they just have a tiny bit of bias, and asks if you are interested. Should you buy the stream from Alice and use it to encrypt your most sensitive data?

Of course, this example is made up—no one goes to the mall anymore—but the idea generalizes. How bad can slightly biased randomness be?

Problem Oracle. We are given an oracle that takes a `message` and returns

`PRG(K,nonce)⊕(message||FLAG)`

and the goal is to retrieve the secret `FLAG`.

Right now, nothing seems wrong with this oracle. The oracle uses a fresh `nonce` for each query and the `PRG` brand evokes a sense of trust in the quality of the randomness. But that is a façade, the `PRG` is actually biased. In each `PRG` output there is a byte at a fixed position $i \in [0, 20)$ the output of which is biased, i.e., it is not sampled uniformly at random[1].

The goal is to figure out which output position is biased, and then use that bias to recover the `FLAG`. First, you can figure out the bias by sending a bunch of queries with the same message, storing the outputs in an array, and then analyzing it. Second, once you know the position of the biased bit, you can vary the `message` length to position bytes of the `FLAG` one-by-one over the biased position to recover it.

Query Estimates. This attack should ideally take a few hundred queries. If you're querying a lot more than that (say 10s or 100s of thousands), you might be want to update your attack.

Flag Format. `Bias{[2 random bytes in hex]}`, like `Bias{ffff}`.

What to Submit.

- (2 pts) *Source code.* Copy the file `hw1_recover_bias.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw1_recover_bias.py` to CMSX.
- (3 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in text (not hex) in a new text file `hw1_flag_bias.txt` and submit it to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:
  1. (3 pts) Explain how you found the flag?
  2. (1 pts) Estimate the number of queries your attack issues per byte of `FLAG`?
  3. (1 pts) What are the limitations of this attack? Specifically, what is the maximum number of `FLAG` bytes you can recover?

[1] Note that the output of `PRG` has the same length as `message||FLAG`, but the biased position is always the same.

Question 2. ECB Partial Plaintext Recovery

Introduction. As we saw in class, if you encrypt with ECB, you still see penguins. But can you do more? Can you decrypt a ciphertext? Let's find out.

Problem Oracle. We are given an oracle that takes a `message` and returns
`AES128-ECB-Encrypt(K, PKCS7Padding(message||FLAG))`
and the goal is to retrieve the secret `FLAG`.

Don't be alarmed by the `PKCS7Padding`. Recall that ECB mode can only encrypt messages whose length is a multiple of the block size. `PKCS7Padding` is a simple way of padding messages, so they reach that length. It first checks how many bytes one needs to add to the plaintext for its length to be a multiple of the block size, say this number is `4`, then it adds padding of `0x04 0x04 0x04 0x04`. Making the padding bytes the amount makes it easy to remove the padding, you just read the last byte and then remove that many bytes from the end. And in the edge case where the message is already padded, it adds a block of padding `0x10 ... 0x10`, so it remains unambiguous.

The padding part seems like an implementation detail, so why did I spell it out. It turns out to be critical to this attack. More generally, in computer security, it is useful to keep an eye out for stuff that other folks think is "trivial" or "obvious" because that is likely where they messed up and opened themselves up to attacks.

So, how can we leverage padding to recover the flag? First, notice that by querying messages of different lengths, we can uncover the length of the `FLAG`. So, if we query a message whose length when added to the length of the `FLAG` is a multiple of the block size, then the entire last block (after padding but before encryption) will be `0x10 ... 0x10`. That's a good start, we don't know what the `FLAG` bytes are, but we know what comes next. Now, if we send a message that is slightly longer such that one byte of the `FLAG` overflows into the next block, the last block would now look like

```
[last byte of FLAG] 0x0f 0x0f ... 0x0f
```

This is amazing, we know all the bytes except one, and a byte can only have 256 possible values and brute forcing it, we can find out the last byte of `FLAG`. Then we can incremenet the length of our message by another byte and brute-force the second last byte of `FLAG`.

```
[second last byte of FLAG] [last byte of FLAG (known!)] 0x0e 0x0e ... 0x0e
```

And we can continue this process till we recover all the bytes of `FLAG`.

Query Estimates. This attack should ideally take a thousand queries. If you're querying a lot more than that (say 10s or 100s of thousands), you might be want to update your attack.

Query Length Limits. It is possible (but not necessary!) to do optimizations that reduce the number of queries at the cost of longer queries (like submitting a query that tries all the 256 values of a byte at once). We encourage you to experiment with such optimizations, but be mindful that you may run into our query length limit of 200,000 bytes.

Flag Format. `ECB{[some random bytes in hex]}`, like `ECB{ff...ff}`.

What to Submit.

- (2 pts) *Source code.* Copy the file `hw1_recover_ecb.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw1_recover_ecb.py` to CMSX.
- (3 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in text (not hex) in a new text file `hw1_flag_ecb.txt` and submit it to CMSX.
- (5 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:

    1. (3 pts) Explain how you found the flag?
    2. (2 pts) Estimate the number of queries your attack issues to recover the `FLAG`?

Question 3. CTR Plaintext Recovery

Introduction. CTR mode forms the basis for modern symmetric encryption, it underlies both the symmetric encryption algorithms in TLS, AES-GCM and ChaCha20/Poly1305. When implemented correctly, it provides confidentiality against passive adversaries and is incredibly fast.

3.1. Implement CTR given AES

This sub-problem is slightly different from previous problems. Here, we'd like you to implement CTR mode encryption given access to the AES block cipher.

We have provided you with a starter template in `hw1_AESCtr.py` in the `code` directory. You can start by using that and furnishing the TODOs. Of course, since the goal of this is to get you to implement CTR mode, you are not allowed to use libraries (like pycryptodome) that already implement CTR mode.

We have also included a basic test file `test_hw1_AESCtr.py`, which you can invoke by running `poetry run pytest test_hw1_AESCtr.py` in the homeworks directory (as discussed in the setup portion earlier.)

What to Submit.

- (4 pts) *Source code.* Copy the file `hw1_AESCtr.py` into your homeworks' folder and complete the TODOs to implement CTR. Submit `hw1_AESCtr.py` to CMSX and to Gradescope as well.

3.2. Break a given CTR mode ciphertext with nonce-reuse

The "implemented correctly" probably gave it away but if you implement CTR mode incorrectly, it can be horrendously insecure. In this sub-problem, we will look at a specific implementation error and see how we can attack it.

Problem Oracle. We are given an oracle that takes a `message` and returns
`AES128-CTR-Encrypt(K, nonce, message||FLAG)`
and the goal is to retrieve the secret `FLAG`.

This looks secure, but the oracle has an implementation error where the `nonce` is fixed across queries.

Think about how the nonce is used in CTR mode and how keeping it constant may leak information.

Query Estimates. This attack should ideally take a handful of queries. If you're querying a lot more than that (say tens or hundreds), you might be want to update your attack.

Flag Format. `CTR{[2 random bytes in hex]}`, like `CTR{ffff}`.

What to Submit.

- (1 pts) *Source code.* Copy the file `hw1_recover_ctr.py` into your homeworks' folder and complete the TODOs to conduct the attack. Submit `hw1_recover_ctr.py` to CMSX.
- (2 pts) *Captured flag.* Put the captured `FLAG`, just the `FLAG` no quotes or special characters in

text (not hex) in a new text file `hw1_flag_ctr.txt` and submit it to CMSX.

- (3 pts) *Written answer.* Submit your answers to the following questions to Gradescrope:
  1. (2 pts) Explain how you found the flag?
  2. (1 pts) Estimate the number of queries your attack issues to recover `FLAG`?

Evaluation

We would love it if you could provide us feedback to improve future homeworks. Please provide answers to the following questions on Gradescrope.

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours did you spent on this homework?
- Did you feel that the coding was too much, appropriate, or not enough?

And feel free to include additional feedback (on the homework or logistics in general). Also, this is not a tight deadline, we'd be delighted to hear your feedback via Slack before or after the deadline.